APPENDIX A

DIGITAL SIGNAL PROCESSING WITH EFFICIENT POLYPHASE RECURSIVE ALL-PASS FILTERS

fred harris*, Maximilien d'Oreye de Lantremange*, and A.G. Constantinides**

* Department of Electrical and Computer Engineering, San Diego State University, San Diego, California 92182-0190, USA.
** Signal Processing Section, Department of Electrical Engineering, Imperial College of Science, Technology and Medicine, Exhibition Road, London SW7-2BT, England.

ABSTRACT: Digital filters can be realized with remarkably small computational burden as polyphase recursive all-pass networks. These networks are formed as a sum of M parallel all-pass subfilters with phase shifts selected to add constructively in the passband and destructively in the stopband. The design technique we present here starts with prototype M-path recursive all-pass filters obtained by a new optimizing algorithm (described in detail elsewhere). These filters are operated with combinations of all-pass transformations, resampling, and cascading options. We demonstrate a number of designs using the new algorithm along with examples of filtering systems obtained by the techniques presented.

1. INTRODUCTION

A standard finite impulse response (FIR) filter can be modeled, as indicated in Fig. 1, as the weighted summation of the contents of a tapped delay line. This summation is shown in (1) and the transfer function of this filter is shown in (2).

$$y(n) = \sum_{m=0}^{M-1} h(m-n)\, x(m) \qquad (1)$$

$$Y(Z) = X(Z) \sum_{n=0}^{M-1} h(n)\, Z^{-n} \qquad (2)$$

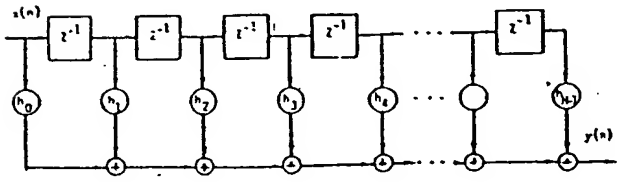

FIGURE 1. TAPPED DELAY LINE FIR FILTER

We note the frequency selective characteristics of the filter output is due to the phase shift between successive samples in the tapped delay line. This phaser summation is shown in Fig. 2 for two different frequencies.

Note that it is the phase shift, and not the weighting terms which is responsible for the frequency dependent gain of the filter. The weighting terms are used primarily to control passband width and stopband attenuation of the filter. An idea derived from spatial beamforming (known as beam spoiling) [1] and from signal synthesis (peak-to-rms control) [2] is to use structured phase shifts (with arbitrary amplitudes) to accomplish the same goal.

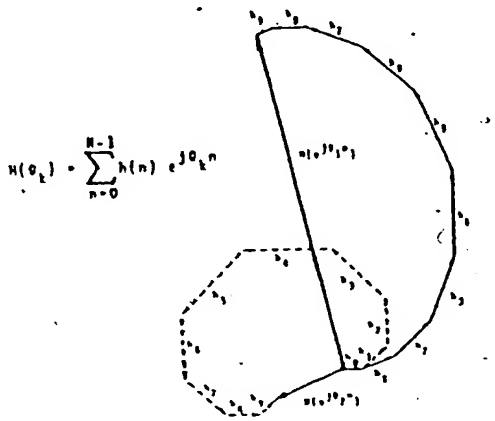

FIGURE 2. PHASOR SUMMATION OF FIR FILTER OUTPUT FOR TWO FREQUENCIES

The class of filters we describe here replaces the weights of the FIR filter with all-pass subfilters which exhibit unity gain with frequency dependent phase shift. In anticipation of the polyphase structure, the all-pass subfilters are, as shown in (3) and Fig. 3, first order polynomials in Z^{-M} where M is the number of taps in the structure. This composite form is indicated in Fig. 4a with a modified form reflecting a polyphase structure in 4b. The transfer function of this structure is shown in (4).

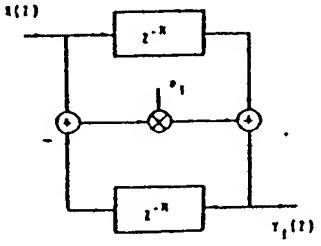

FIGURE 3. M-TH ORDER ALL-PASS SUBFILTER

Presented at International Conference on Signal Processing
Florence, Italy, Sept. 4-6 1991.

Fig. 7 show examples of equal ripple designs obtained from this new algorithm for a 2-path and for a 5-path filter with two and three all-pass stages per path respectively. Interacting constraints on the optimal design (described in the aforementioned paper) limit the possible number of stages per path for the 5-path filter to the sequences (1,1,1,1,0), (2,2,2,1,1), (3,3,2,2,2), etc., so that the optimum 5-path filter uses three less stages than allocated to the design. Similarly the 2-path filter is limited to the sequences (1,0), (1,1), (2,1), (2,2), (2,3), (3,3), etc.

2. TWO PATH FILTERS

We find a wealth of interesting properties and clever applications for this structure even if we limit our discussion to a 2-path filter. As designed, the 2-path filter is a half bandwidth filter with the 3-dB bandwidth at 0.25 f_s. If the zeros are restricted to the half sampling frequency, the filter identical to a half bandwidth Butterworth obtained by the standard warped bilinear transform. For this special case, the real parts of the root locations go to zero. If the zeros are optimized for equal ripple stop band behavior the filter becomes a constrained Elliptic filter. The constraints are related to a property of complementary all-pass filters. We define the all-pass sections for the 2-path filter as $H_0(\theta)$ and $H_1(\theta)$ and, as indicated in Fig. 8, the scaled by one-half sum and difference of these paths by $A(\theta)$ and $B(\theta)$, the lowpass and highpass filters, respectively. We know that the all-pass sections satisfy (5) from which we derive the power relationship between the lowpass and highpass filters shown in (6).

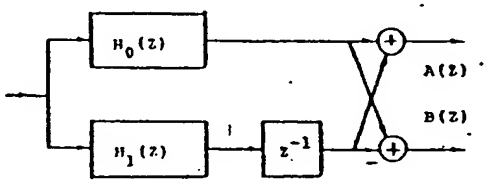

FIGURE 8. COMPLEMENTARY LOWPASS AND HIGHPASS FILTERS FROM 2-PATH ALL-PASS NETWORK

$$|H_0(\theta)|^2 = |h_1(\theta)|^2 = 1 \qquad (5)$$

$$|A(\theta)|^2+|B(\theta)|^2 = |0.5[H_0(\theta)+H_1(\theta)]|^2 \qquad (6a)$$

$$+ |0.5[H_0(\theta)-H_1(\theta)]|^2$$

$$= 0.5[|H_0(\theta)|^2+|H_1(\theta)|^2] \qquad (6b)$$

$$= 1 \qquad (6c)$$

Now to interpret how this relationship impacts the 2-path filter. For the complementary filters we define the minimum passband gain of by $1-\epsilon_1$ and the peak stopband gain by ϵ_2. Substituting these gains in (6) we obtain (7).

$$(1-\epsilon_1)^2 + (\epsilon_2)^2 = 1 \qquad (7a)$$

$$(1-\epsilon_1)^2 = 1 - (\epsilon_2)^2 \qquad (7b)$$

$$1 - 2\epsilon_1 + (\epsilon_1)^2 = 1 - (\epsilon_2)^2 \qquad (7c)$$

For small ϵ_1, we can ignore $(\epsilon_1)^2$ which leads to (8).

$$\epsilon_1 = 0.5\ (\epsilon_2)^2 \qquad (8)$$

Thus if the stopband attenuation is selected to be 0.001 (60.0 dB), the passband ripple is 0.000 000 5 (126.0 dB) or the passband minimum gain is 0.999 999 5 (-0.000 000 22 dB). As can be seen these filters have a remarkably flat passband. We conclude that the 2-path equal ripple stopband filters are elliptic filters with coupled passband and stopband ripple with -3.0 dB gain at 0.25 f_s. There are two significant differences between the 2-path and the standard implementation of the equivalent elliptic filter. The first is a four-to-one savings in multiplications; a fifth order half bandwidth 2-path filter requires only two coefficients as opposed to the direct implementation which requires eight (scaling included). The second is that all-pass structures exhibit unity gain to internal states hence do not require extended precision registers to store internal states as do standard implementations.

As with any filter design, the transition bandwidth can be traded for out of band attenuation for a fixed order or transition bandwidth can be steepened for a fixed attenuation by increasing filter order. While nomographs exist for the elliptic and Butterworth filters which can be implemented by the 2-path structure, a simple approximate relationship for the equal ripple filter is given in (9), where A(dB) is attenuation in dB, Δf is transition bandwidth, and N is the total number of all-pass segments in the filter.

$$A(dB) = (72\ \Delta f + 10)\ N \qquad (9)$$

2A. HILBERT TRANSFORM FILTERS

The Hilbert Transform (HT) can be thought of as a particular type of half band filter. It is modelled and implemented by a wide band 90° phase shifting network. The HT is often used in DSP applications to form the analytic signal a(n) as shown in (10), a signal with spectra limited to the positive (or negative) frequency band.

$$a(n) = x(n) + j\ \hat{x}(n) \qquad (10)$$

We can cast the 2-path filter as a HT in a number of ways but a simple method, akin to the transformation performed on a half band FIR filter [4], is a heterodyne of the half band filter to the quarter sample frequency. If the original filter response and transform are denoted by h(n) and H(Z) respectively, then the heterodyned expressions can be easily seen to be those in (11).

$$h(n) \Rightarrow h(n)\ e^{jm\pi/2} = h(n)\ (j)^n \qquad (11a)$$

$$H(Z) \Rightarrow H(Z\ e^{j\pi/2}) = H(j\ Z) \qquad (11b)$$

The transformation of a half band filter to an HT filter is accomplished by replacing each Z^{-1} of the transfer function with $-jz^{-1}$. Since the polynomials of the all-pass subfilters are second order, this transfor-

mation can be accomplished by changing the sign of the coefficient used in each all-pass sub filter and by associating a -j with the lower path delay. These operations are indicated in Fig. 9.

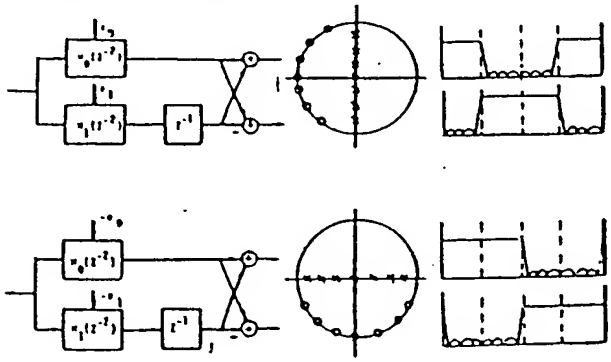

FIGURE 9. HILBERT TRANSFORM ALL-PASS FILTER REALIZATION AS TRANSFORMED HALF BAND FILTER

2B. INTERPOLATION AND DECIMATION FILTERS

Any half bandwidth filter can be used to perform 2-to-1 resampling (up or down, popularly identified as interpolation and decimation). Finite impulse response (FIR) filters can be operated with a polyphase partition which does not process inserted zero input points (up-sampling) or compute the discarded output points (down sampling). While this option is not available for the general recursive filter, it is an option for the recursive all-pass M-path filter. The M-path filter can be partitioned into polyphase segments because of the interaction of the internal delays of the M-th order subfilters and the delay line of each path. This relationship is particularly easy to see for the 2-path filter. An impulse applied at index n_0 makes a contribution to the output at the same time via the upper path while an output is not available from the lower path till the next index due to its extra delay Z^{-1}. During that same next index, the upper path offers no contribution to the output because there is no Z^{-1} path associated with its Z^{-2} polynomial. Thus the filter delivers successive samples of its impulse response from alternate paths of the filter.

Down sampling is accomplished by delivering alternate input samples to each path of the filter operating at the reduced input rate. Since the delay of the second path is accomplished by the input commutation the physical delay element in that path is removed. The filtered and down sampled out-put is generated by the sum or difference of the two path responses. Note that the computation rate per input sample point is half that of the non resampled filter, thus if the 2-path filter of Fig. 7 is used for down sampling we would perform 3 multiplications per input point.

Up sampling is accomplished by reversing the down sampling process. This entails delivering the same input to the two paths and commutating between their outputs. Both forms of resampling are shown in Fig. 10. As in the previous example, when the 2-path filter of Fig. 7. is used for up sampling we would perform 3 multiplications per output point.

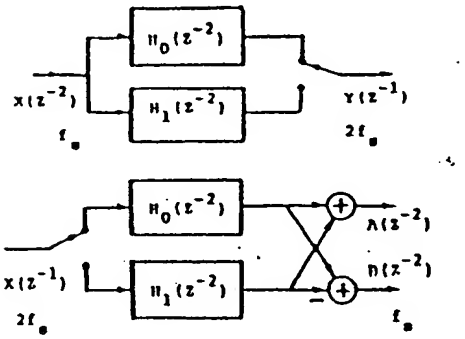

FIGURE 10. RESAMPLING 2-PATH ALL-PASS FILTER

The cascade operation of down sampling and up sampling of the complementary outputs performs two band maximally decimated quadrature mirror filtering and reconstruction [5].

Cascading multiple stages of up sampling or down sampling filters can achieve high order sample rate change with remarkably small workload per data point. For instance, a 1-to-16 up sampler with 96 dB dynamic range for a signal with cutoff frequency of 0.4 f_s requires 2-path filters with successively smaller number of all-pass stages of orders (3,3), (2,2), (2,1), and (1,1). This requires a total of 42 multiplications for 16 outputs, for an average workload of approximately 2.7 operations per output point.

2C. ITERATED POLYPHASE ALL-PASS FILTERS

The all-pass networks described earlier are formed by polynomials in Z^{-M} and specifically for the 2-path filters Z^{-2}. A useful transformation is to replace each Z^{-2} with Z^{-2K}, to obtain higher order filters exhibiting K-fold spectral replication of the original filter. For K=2, the half band filter centered at DC becomes a pair of quarter band filters centered at DC and $f_s/2$. An example of spectral replication for the iterated filter is shown in Fig. 11.

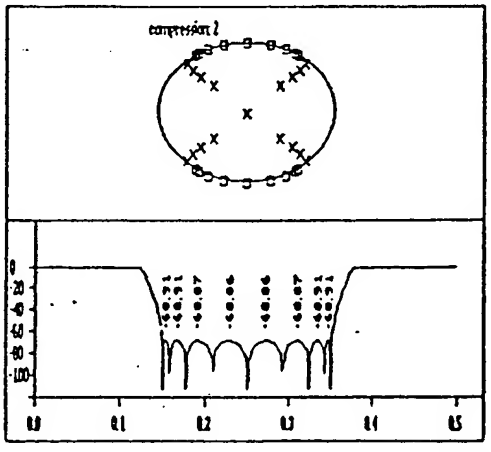

FIGURE 11. SPECTRUM OF ITERATED 2-PATH FILTER: POLYNOMIALS IN Z^{-4}

Note that this transformation scales the prototype filter's spectrum by 1/K, reducing both passband width and transition width. Thus very steep transition bandwidths and narrow filters can be realized by the use of low order polynomials of degree 2K obtained via delay elements. Energy in the replicated spectral regions can be removed by a sequence of filters incorporating various degrees of resampling and iterated filters of reduced degree.

3. EFFICIENT ALL-PASS FILTER BANKS.

Efficient spectral partitioning can be obtained by cascading and resampling the outputs of iterated lowpass and HT filters. For instance the spectra of a four channel filter bank centered on the four quadrants can be formed with a complementary half band filter followed by a pair of resampled HT filters as shown on the left side of Fig. 12. The workload for this 70 dB attenuation filter set is 3 multiplications per output channel.

A four channel filter bank straddling the previous set (ie. centered on the four cardinal directions) can be formed with an iterated by two complementary half band filter followed by both a complementary half band resampled filter and a resampled HT filter. This spectra of this filter set is shown on the right side of Fig. 12. The workload for this 70 dB attenuation filter set is 2 multiplications per output channel.

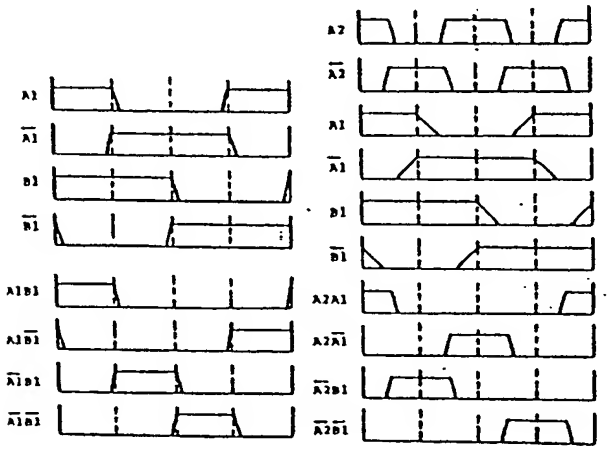

FIGURE 12. SPECTRA OF FOUR CHANNEL QUADRANT CENTERED AND CARDINAL CENTERED FILTER BANK

4. OTHER ALL-PASS TRANSFORMATIONS

The prototype complementary all-pass filter set can also be transformed to filters of arbitrary bandwidth and of arbitrary center frequency by standard all-pass transformations [6]. The lowpass transformation for the half band filter is shown in (12).

$$z^{-1} \Rightarrow \frac{b + z^{-1}}{1 + b\, z^{-1}}, \qquad b = \frac{1-\tan(\theta_0/2)}{1+\tan(\theta_0/2)} \qquad (12)$$

This transformation warps the frequency variable of the filter so that the -3 dB point resides at frequency θ_0. When θ_0 is different from $\pi/2$ the second order polynomials of the all-pass subfilters acquire coefficients for the z^{-1} term. The resulting filter requires two multiplications per pole zero pair which is still half the workload of the traditional canonic forms. In addition, the pole at the origin (due to the delay of the second path) transforms to a first order all-pass filter thus converting an inactive pole to an active one requiring an additional first order stage. Fig. 13 presents the frequency warped version of the filter presented in Fig. 7a. Both this filter and its complement are available from the warped structure.

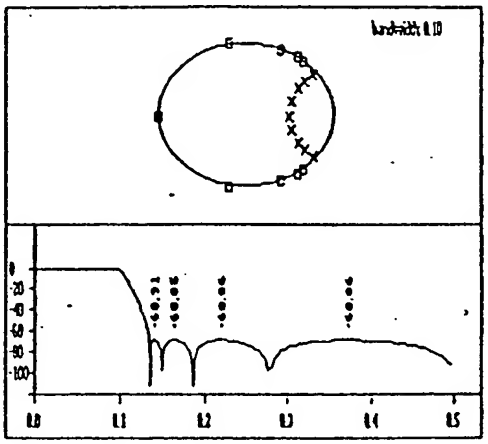

FIGURE 13. LOWPASS TRANSFORMATION OF ALL-PASS FILTER

The bandpass transformation is presented in (13).

$$z^{-1} \Rightarrow -z^{-1}\ \frac{c - z^{-1}}{1 - c\, z^{-1}}, \quad c = \cos(\theta_1) \qquad (13)$$

This transformation warps the frequency variable of the filter so that the center frequency resides at frequency θ_1. If θ_1 is equal to 0.0 this transformation is equivalent to the iterated transformation described in 2.C. For any other frequency, this transformation converts the second order all-pass filters to fourth order all-pass structures. This requires four multiplications to form four pole-zero pairs of the resultant filter which is still half the workload of the traditional canonic forms. In addition, the transformation converts the real pole (which is an image of the pole originally at the origin) to a pair of complex poles requiring a second order all-pass stage. Fig. 14 presents the frequency warped version of the filter presented in Fig. 14. As earlier, both this filter and its complement are available from the warped structure.

Other transformations and geometries [7] can be used with the digital all-pass structure and the reader is directed to the bibliography of the listed papers for a profusion of options.

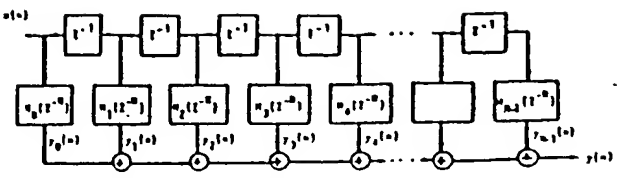

FIGURE 4A. ALL-PASS FILTER STRUCTURE

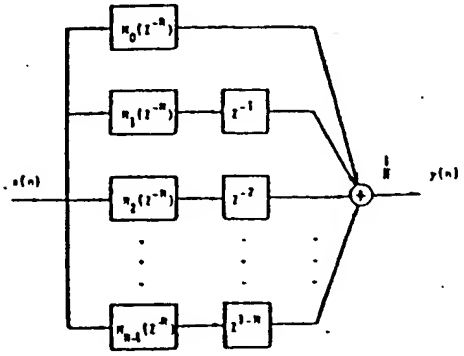

FIGURE 4B. POLYPHASE ALL-PASS STRUCTURE

$$H_n(Z^{-M}) = \frac{a_n + Z^{-M}}{1 + a_n Z^{-M}} \qquad (3)$$

$$Y(Z) = X(Z) \sum_{n=0}^{M-1} H_n(Z^{-M})\, Z^{-n} \qquad (4)$$

The roots of (3), the M-th roots of $-a_n$ and its reciprocal, are equally spaced about the origin as shown in Fig. 5 for the indicated degrees M. Note that we realize M poles and M zeros per multiplication in the all-pass stage. This means for instance that a 2-path filter offers two poles and two zeros per multiplication as opposed to one pole (or zero) per multiplication for the standard (factored or unfactored) canonic forms.

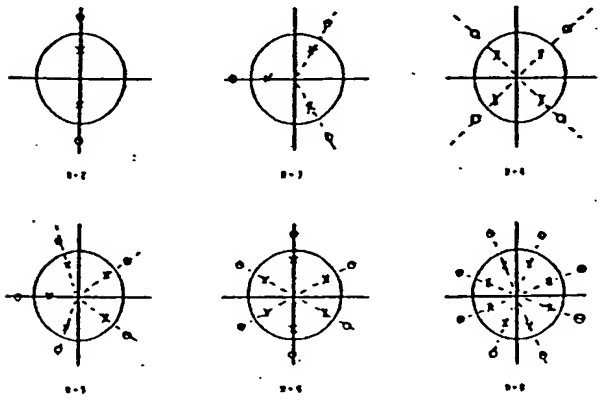

FIGURE 5. POLE-ZERO DISTRIBUTION FOR M-TH ORDER ALL-PASS SUBFILTERS

As we go around the unit circle we visualize that the all-pass subfilters exhibit rapid change in phase in the neighborhood of each pole-zero pair. By proper choice of the pole positions, the phase shift of each leg of the M path filter can be made to match or to differ by multiples of $2\pi/M$ over selected spectral intervals. This is suggested in Fig. 6 for M=2 and M=4. The coefficients of the all-pass subfilters can be determined by standard algorithms [3] or by a new algorithm developed by harris and d'Oreye and reported in a paper recently submitted for publication (Signal Processing).

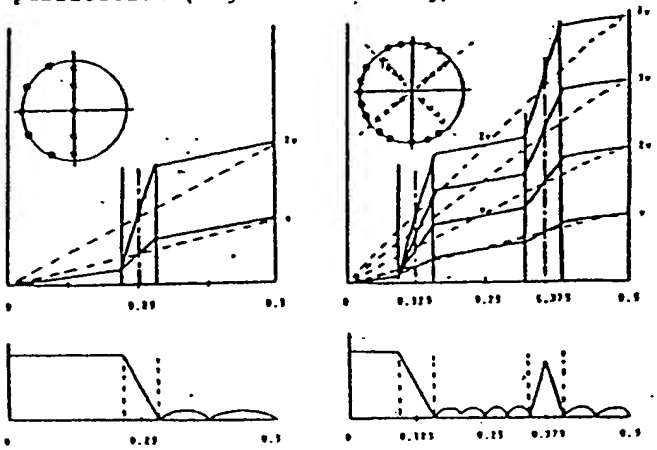

FIGURE 6. TYPICAL PHASE RESPONSES OF ALL-PASS FILTER PATHS, AND CORRESPONDING MAGNITUDE RESPONSES.

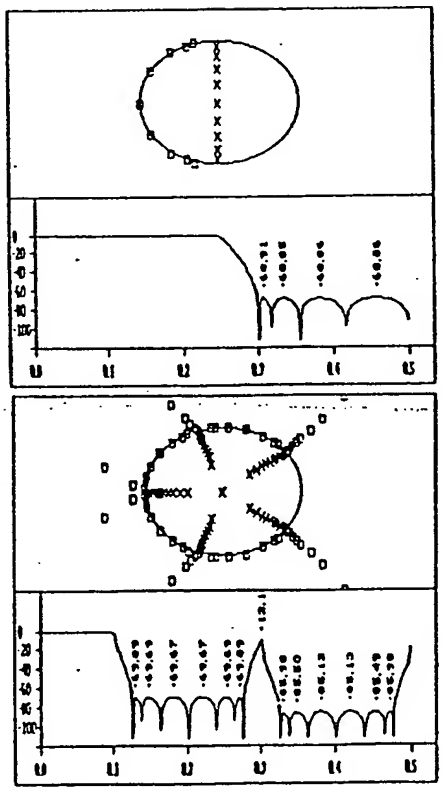

FIGURE 7. POLE-ZERO PLOT AND MAGNITUDE RESPONSE FOR 2-PATH AND 5-PATH POLYPHASE ALL-PASS NETWORK

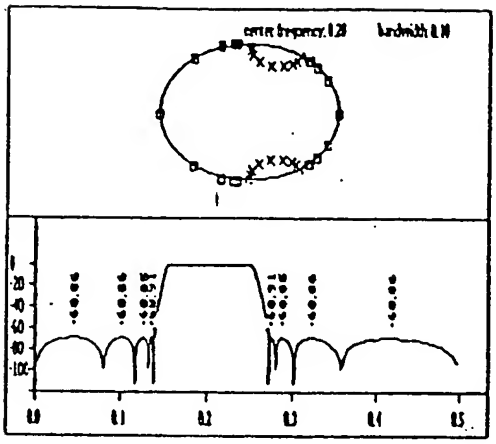

FIGURE 14. BANDPASS TRANSFORMATION OF ALL-PASS FILTER

5. CONCLUSIONS

We have reviewed the form and utility of the all-pass polyphase filter structure. We then presented a number of all-pass transformations which could be easily applied to M-path all-pass filter sets. The emphasis has been on 2-path networks but the material is easily extended to arbitrary M. We alluded to a new set of algorithms for designing M-path filters which will be reported in an upcoming Signal Processing paper. We have demonstrated capabilities of this algorithm by using it to generate the examples presented in this paper.

The primary advantage of these structures is very low workload for a given filtering task. Other papers [8,9,10] have discussed the low sensitivity to finite arithmetic exhibited by these filters. The primary impediment to the use of these filters is their newness and the lack of available design methods for computing filter weights. This paper (along with excellent surveys listen in the bibliography) address the first problem and a forthcoming paper addresses the second.

6. ACKNOWLEDGEMENT

This work was sponsored in part by the Industry/University Cooperative Research Center (I/UCRC) on Integrated Circuits and Systems (ICAS) at San Diego State University (SDSU) and the University of California, San Diego (UCSD).

7. BIBLI GRAPHY

1. Private correspondence between f.j. harris and Stan Acks of Hughes Aircraft, Radar Systems Group, El Segundo, CA. 1983.

2. M.R. Schroeder, "Number Theory in Science and Communications", Springer-Verlag, 1984, Chapter 28, Waveforms and Radiation Patterns, pp. 278-288.

3. R.A. Valenzuela and A.G. Constantinides, "Digital Signal Processing Schemes for Efficient Interpolation and Decimation", IEE Proceedings, Vol. 130, Pt. G. No. 6, Dec. 1983, pp. 225-235.

4. D. Elliot, "Handbook of Digital Signal Processing, Engineering Applications", Academic Press, 1987, Chapter 3, pp. 227-233.

5. Phillip A. Regalia, Sanjit K. Mitra, and P.P. Vaidyanathan, "The Digital All-Pass Filter: A Versatile Signal Processing Building Block", Proc. of IEEE, Vol. 76, No. 1, Jan. 1988, pp. 19-37.

6. A.G. Constantinides, "Spectral Transformations for Digital Filters", Proc. IEE, Vol. 117, No. 8, Aug. 1970, pp. 1585-1590.

7. S.K. Mitra, K. Hirano, S. Nishimura, and K. Sugahara, "Design of Digital Bandpass-/Bandstop Filters with Independent Tuning Characteristics", FREQUENZ, 44(1990)3-4, pp. 117-121.

8. R. Ansari and B. Liu, "A Class of Low-Noise Computationally Efficient Recursive Digital Filters with Applications to Sampling Rate Alterations", IEEE Trans. Acoust., Speech, Signal Processing, Vol. ASSP-33, No. 1. Feb. 1985, pp. 90-97.

9. H. Samueli, "A Low-Complexity Multiplier-less Half-Band Recursive Digital Filter Design", IEEE Trans. Acoust., Speech, Signal Processing, Vol. ASSP-37, No. 3. Mar. 1989, pp. 442-444.

10. P.P. Vaidyanathan and Zinnur Doganata, "The Role of Lossless Systems in Modern Digital Signal Processing: A Tutorial", IEEE Trans. on Education, Vol. 32, No. 3, Aug. 1989, pp. 181-197.

```
/*****************************************************************************
*************************** APPENDIX B ***************************************
************************** Least Square Lattice ******************************
************************      Noise Cancelling     ******************************/
/* Example for ratiometric approach to noise cancelling */
#define LAMBDA  0.95

void OxiLSL_NC( int     reset,
                int     passes,
                int     *signal_1,
                int     *signal_2,
                int     *signal_3,
                int     *target_1,
                int     *target_2) {

        int     i, ii, k, m, n, contraction;
static  int     *s_a, *s_b, *s_c, *out_a, *out_c;
static  float   Delta_sqr, scale, noise_ref;


 if( reset == TRUE){
   s_a    = signal_1;
   s_b    = signal_2;
   s_c    = signal_3;
   out_a  = target_1;
   out_c  = target_2;
   factor = 1.5;
   scale  = 1.0 /4160.0;

 /* noise canceller initialization at time t=0 */

   nc[0].berr  = 0.0;
   nc[0].Gamma = 1.0;

   for(m=0; m<NC_CELLS; m++) {
     nc[m].err_a   = 0.0;
     nc[m].err_b   = 0.0;
     nc[m].Roh_a   = 0.0;
     nc[m].Roh_c   = 0.0;
     nc[m].Delta   = 0.0;
     nc[m].Fswsqr  = 0.00001;
     nc[m].Bswsqr  = 0.00001;
   }
}

/*========================== END INITIALIZATION ==============================*/

for(k=0; k<passes; k++){

  contraction = FALSE;
  for(m=0; m< NC_CELLS; m++) {          /* Update delay elements        */
    nc[m].berr1   =  nc[m].berr;
    nc[m].Bswsqr1 =  nc[m].Bswsqr;
  }


  noise_ref    = factor * log(1.0 - (*s_a) * scale)
               - log(1.0 - (*s_b) * scale) ;
  nc[0].err_a = log(1.0 - (*s_b) * scale);
  nc[0].err_b = log(1.0 - (*s_c) * scale);
```

```
++s_a;
++s_b;
++s_c;

nc[0].ferr   = noise_ref ;
nc[0].berr   = noise_ref ;
nc[0].Fswsqr = LAMBDA * nc[0].Fswsqr + noise_ref * noise_ref;
nc[0].Bswsqr = nc[0].Fswsqr;

/* Order Update      */
for(n=1;( n < NC_CELLS) && (contraction == FALSE); n++) {

  /* Adaptive Lattice Section */

  m = n-1;
  ii= n-1;

  nc[m].Delta *=  LAMBDA;
  nc[m].Delta +=  nc[m].berr1 * nc[m].ferr  / nc[m].Gamma ;
  Delta_sqr    =  nc[m].Delta * nc[m].Delta;

  nc[n].fref   = -nc[m].Delta / nc[m].Bswsqr1;
  nc[n].bref   = -nc[m].Delta / nc[m].Fswsqr;

  nc[n].ferr   =  nc[m].ferr  + nc[n].fref * nc[m].berr1;
  nc[n].berr   =  nc[m].berr1 + nc[n].bref * nc[m].ferr;

  nc[n].Fswsqr =  nc[m].Fswsqr  - Delta_sqr / nc[m].Bswsqr1;
  nc[n].Bswsqr =  nc[m].Bswsqr1 - Delta_sqr / nc[m].Fswsqr;

  if( (nc[n].Fswsqr + nc[n].Bswsqr) > 0.00001 || (n < 5) ) {
    nc[n].Gamma = nc[m].Gamma - nc[m].berr1 * nc[m].berr1 / nc[m].Bswsqr1;
    if(nc[n].Gamma < 0.05) nc[n].Gamma = 0.05;
    if(nc[n].Gamma > 1.00) nc[n].Gamma = 1.00;

  /* Joint Process Estimation Section */

    nc[m].Roh_a *= LAMBDA;
    nc[m].Roh_a += nc[m].berr * nc[m].err_a  / nc[m].Gamma ;
    nc[m].k_a    = nc[m].Roh_a  / nc[m].Bswsqr;
    nc[n].err_a  = nc[m].err_a  - nc[m].k_a * nc[m].berr;

    nc[m].Roh_c *= LAMBDA;
    nc[m].Roh_c += nc[m].berr * nc[m].err_b  / nc[m].Gamma ;
    nc[m].k_c    = nc[m].Roh_c  / nc[m].Bswsqr;
    nc[n].err_b  = nc[m].err_b  - nc[m].k_c * nc[m].berr;

  }
  else {
    contraction = TRUE;
    for(i=n; i<NC_CELLS; i++) {
      nc[i].err_a   = 0.0;
      nc[i].Roh_a   = 0.0;
      nc[i]. rr_b   = 0.0;
      nc[i].Roh_c   = 0.0;
      nc[i].Delta   = 0.0;
      nc[i].Fswsqr  = 0.00001;
      nc[i].Bswsqr  = 0.00001;
      nc[i].Bswsqr1 = 0.00001;
```

```
        }

      }

    }

    *out_a++ = (int)( (-exp(nc[ii].err_a) +1.0) / scale) ;
    *out_c++ = (int)( (-exp(nc[ii].err_b) +1.0) / scale) ;


  }
}
/************************    Least Square Lattice    ******************************
***************************                          ********************************
*******************************************************************************/
```

```
/* Normalized Adaptive Noise Canceler   */

#include <stdlib.h>
#include <math.h>

#define TRUE            1
#define FALSE           0

#define FLOAT32         float
#define INT32           long int
#define VOID            void

#define FABSF           fabs
#define SQRTF           sqrt

#define MAX(a,b) (a) > (b) ? (a) : (b)
#define MIN(a,b) (a) < (b) ? (a) : (b)

#define MIN_VAL     0.01
#define MAX_DEL     0.999999
#define MIN_DEL    -0.999999
#define MAX_RHO     2.0
#define MIN_RHO    -2.0
#define MIN_BSERR  1E-15

typedef struct {
    FLOAT32     berr;
    FLOAT32     berr_1;
    FLOAT32     delta;
    FLOAT32     err;
    FLOAT32     ferr;
    FLOAT32     gamma;
    FLOAT32     gamma_1;
    FLOAT32     rho;
    FLOAT32     delta_1;
    FLOAT32     Bserr;
    FLOAT32     Bserr_1;
}   LANC_CELLS;

typedef struct {
    INT32       cc;             /* number of cells              */
    FLOAT32     lambda;         /* put in value for lambda      */
    FLOAT32     min_error;      /* parameter                    */
    LANC_CELLS  *cells;         /* point to array of ANC_CELLS  */
}   LANC_Context;

extern LANC_Context *
LANC_Init(
    INT32       num_cells,      /* number of cells              */
    FLOAT32     lambda,         /* lambda param                 */
    FLOAT32     min_error);     /* min error                    */

extern VOID
LANC_Done(
    LANC_Context        *c);

extern VOID
LANC_Reset(
    LANC_Context    *anc);
```

```
extern FLOAT32
LANC_Calc(
    LANC_Context        *anc,    /* input, context handle        */
    FLOAT32             nps,     /* input, noise plus signal     */
    FLOAT32             noise); /* input, noise reference       */

/* The following macros provide efficient access to the lattice */

#define ANC_CELL_SIZE   11

#define xBERR    0
#define xBERR_1  1
#define xDELTA   2
#define xDELTA_1 3
#define xGAMMA   4
#define xGAMMA_1 5
#define xBSERR   6
#define xBSERR_1 7
#define xERR     8
#define xFERR    9
#define xRho    10

#define berr            (*(p + xBERR))
#define P_berr_1        (*(p + xBERR_1 - ANC_CELL_SIZE))
#define P_berr          (*(p + xBERR   - ANC_CELL_SIZE))
#define berr_1          (*(p + xBERR_1))

#define Bserr           (*(p + xBSERR))
#define Bserr_1         (*(p + xBSERR_1))
#define P_Bserr_1       (*(p + xBSERR_1 - ANC_CELL_SIZE))

#define P_delta         (*(p + xDELTA - ANC_CELL_SIZE))
#define delta           (*(p + xDELTA))
#define delta_1         (*(p + xDELTA_1))
#define P_delta_1       (*(p + xDELTA_1 - ANC_CELL_SIZE))

#define err             (*(p + xERR))
#define N_err           (*(p + xERR + ANC_CELL_SIZE))

#define P_ferr          (*(p + xFERR - ANC_CELL_SIZE))
#define ferr            (*(p + xFERR))

#define gamma           (*(p + xGAMMA))
#define P_gamma         (*(p + xGAMMA   - ANC_CELL_SIZE))
#define N_gamma         (*(p + xGAMMA   + ANC_CELL_SIZE))
#define P_gamma_1       (*(p + xGAMMA_1 - ANC_CELL_SIZE))
#define gamma_1         (*(p + xGAMMA_1))

#define rho             (*(p + xRho))



/* ---------------------------------------------------------------
   Name: LANC_Init

   Abstract: Create an ANC context

   --------------------------------------------------------------- */
```

```
extern LANC_Context *
LANC_Init(
    INT32       num_cells,      /* number of cells              */
    FLOAT32     lambda,         /* lambda param                 */
    FLOAT32     min_error) {    /* min error                    */

    LANC_Context        *anc;   /* context                      */

    anc = (LANC_Context *)malloc(sizeof(LANC_Context));
    assert(anc != NULL);
    anc->cc = num_cells;
    anc->lambda = lambda;
    anc->min_error = min_error;
    anc->cells = (LANC_CELLS *)malloc(sizeof(LANC_CELLS) * (num_cells + 2));
    assert(anc->cells != NULL);

    return(anc);
}

/* ---------------------------------------------------------------------
   Name: LANC_Reset

   Abstract: Reset an ANC context

   --------------------------------------------------------------------- */

extern VOID
LANC_Reset(
    LANC_Context *anc) {

    FLOAT32             *p;
    INT32               m;

    p = (FLOAT32 *)anc->cells;
    for (m = 0; m <= anc->cc; m++) {
        rho     = 0.0;
        err     = 0.0;
        ferr    = 0.0;
        berr    = 0.0;
        berr_1  = 0.0;
        delta   = 0.0;
        delta_1 = 0.0;
        Bserr   = anc->min_error;
        Bserr_1 = anc->min_error;
        gamma   = MIN_VAL;
        gamma_1 = MIN_VAL;
        p      += ANC_CELL_SIZE;
    }
    p = (FLOAT32 *)anc->cells;          /* Cell # 0 special case    */
    gamma   = 1.0;
    gamma_1 = 1.0;
}


/* ---------------------------------------------------------------------
   Name: LANC_Done

   Abstract: Delete an ANC context

   --------------------------------------------------------------------- */
```

```
extern VOID
LANC_Done(
    LANC_Context        *anc) {

    free(anc->cells);
    free(anc);
}

/* ------------------------------------------------------------------
   Name: LANC_Calc

   Abstract: Calculate

   ------------------------------------------------------------------ */

FLOAT32
LANC_Calc(
    LANC_Context        *anc,     /* input, context handle        */
    FLOAT32             nps,      /* input, noise plus signal     */
    FLOAT32             noise)    /* input, noise reference       */
{
    INT32       m;
    FLOAT32     *p;
    FLOAT32     B,F,B2,F2;
    FLOAT32     qd2,qd3;
    INT32       output_cell;

    /* Update time delay elements in cell structure ------------- */

    p = (FLOAT32 *)anc->cells;
    for (m = 0; m <= anc->cc; m++) {
        gamma_1 = gamma;
        berr_1  = berr;
        Bserr_1 = Bserr;
        delta_1 = delta;
        p += ANC_CELL_SIZE;
    }

    /* Handle Cell # 0 ------------------------------------------ */
    p = (FLOAT32 *)anc->cells;
    Bserr = anc->lambda * Bserr_1 + noise * noise;
    Bserr = MAX(Bserr, MIN_BSERR);

    ferr  = noise / SQRTF(Bserr);
    ferr  = MAX(ferr, MIN_DEL);
    ferr  = MIN(ferr, MAX_DEL);

    berr  = ferr;

    rho = anc->lambda * SQRTF(Bserr_1 / Bserr) * rho + berr * nps;

    N_err = nps - rho * berr;

    output_cell = anc->cc - 1;      /* Assume last cell for starter */

    for (m = 1; m < anc->cc; m++) {
        p += ANC_CELL_SIZE;

        B  = SQRTF(1.0 - P_berr_1 * P_berr_1);     B2 = 1.0/B;
```

```
        F  = SQRTF(1.0 - P_ferr   * P_ferr );      F2 = 1.0/F;

        P_delta = P_delta_1 * F * B + P_berr_1 * P_ferr;
        P_delta = MAX(P_delta, MIN_DEL);
        P_delta = MIN(P_delta, MAX_DEL);
        qd3   = 1.0 - P_delta * P_delta;
        qd2   = 1.0 / SQRTF(qd3);

        ferr  = (P_ferr    - P_delta * P_berr_1) * qd2 * B2;
        ferr  = MAX(ferr, MIN_DEL);
        ferr  = MIN(ferr, MAX_DEL);

        berr  = (P_berr_1  - P_delta * P_ferr  ) * qd2 * F2;
        berr  = MAX(berr, MIN_DEL);
        berr  = MIN(berr, MAX_DEL);

        gamma = P_gamma * (1.0 - P_berr * P_berr);
        gamma = MAX(gamma, MIN_VAL);
        gamma = MIN(gamma, MAX_DEL);

        Bserr = P_Bserr_1 * qd3;

        Bserr  = MAX(Bserr, MIN_BSERR);

        rho  *= anc->lambda * SQRTF((Bserr_1 / Bserr) * (gamma / gamma_1));
        rho  += berr * err;
        rho   = MAX(rho, MIN_RHO);
        rho   = MIN(rho, MAX_RHO);

        N_err = err - rho * berr;
    }

    p = (FLOAT32 *)&(anc->cells[output_cell /* *ANC_CELL_SIZE */]);
    return(N_err);
}
```

```
/*      QRD                                     */

#include <stdlib.h>
#include <math.h>

#define TRUE            1
#define FALSE           0

#define FLOAT32         float
#define INT32           long int
#define VOID            void

#define FABSF           fabs
#define SQRTF           sqrt


typedef struct {
    INT32       dummy;
} LQRDJPEF_CONTEXT;

typedef LQRDJPEF_CONTEXT * LQRDJPEF_Handle;

extern LQRDJPEF_Handle
LQRDJPEF_Init(
    INT32       NumCells,
    FLOAT32     Lambda,
    FLOAT32     SumErrInit,
    FLOAT32     GamsInit,
    FLOAT32     MinSumErr);

extern VOID
LQRDJPEF_Done(
    LQRDJPEF_Handle hJPE);

extern VOID
LQRDJPEF_Reset(
    LQRDJPEF_Handle hJPE);

extern FLOAT32
LQRDJPEF_Calc(
    LQRDJPEF_Handle hJPE,   /* handle                   */
    FLOAT32  nps,           /* noise plus signal        */
    FLOAT32  noise);        /* noise reference          */



#define MAX(a,b) (a) > (b) ? (a) : (b)
#define MIN(a,b) (a) < (b) ? (a) : (b)


#define LQRDJPEF_CELL_SIZE              27

typedef struct {
         FLOAT32 sinf , sinf_1  ;
         FLOAT32 sinb , sinb_1  ;

         FLOAT32 cosf , cosf_1  ;
         FLOAT32 cosb , cosb_1  ;

         FLOAT32 epsf;
```

```
        FLOAT32 epsb , epsb_1  ;

        FLOAT32 pief , pief_1  ;
        FLOAT32 pieb , pieb_1  ;

        FLOAT32 Fserr, Fserr_1, SQRTF_Fserr, SQRTF_Fserr_1;
        FLOAT32 Bserr_1, Bserr_2, SQRTF_Bserr_1, SQRTF_Bserr_2;

        FLOAT32 p_1, p_2;
        FLOAT32 gams_1;
        FLOAT32 epsi_1;

} LQRDJPEF_CELL;

typedef struct {
        INT32           NumCells;         /* number of cells
        FLOAT32         Lambda;           /* Lambda
        FLOAT32         SumErrInit;       /* Initial value for Fserr, Bserr
        FLOAT32         GamsInit;         /* Initial value for gams
        FLOAT32         MinSumErr;        /* Minimum for Fserr, Bserr
        FLOAT32         SQRTF_Lambda;     /* square root of Lambda
        FLOAT32         SQRTF_SumErrInit;/* square root of SumErrInit
        LQRDJPEF_CELL   *cells;           /* point to array of JPE_CELLS
}   LQRDJPEF_Context;

/*      The following macros provide efficient access to the lattice

        Define variables offsets within structure                        */

#define xSINF           0
#define xSINF_1         1
#define xSINB           2
#define xSINB_1         3

#define xCOSF           4
#define xCOSF_1         5
#define xCOSB           6
#define xCOSB_1         7

#define xEPSF           8
#define xEPSB           9
#define xEPSB_1         10

#define xPIEF           11
#define xPIEF_1         12
#define xPIEB           13
#define xPIEB_1         14

#define xFSERR          15
#define xFSERR_1        16
#define xSQRTF_FSERR    17
#define xSQRTF_FSERR_1  18

#define xBSERR_1        19
#define xBSERR_2        20
#define xSQRTF_BSERR_1  21
#define xSQRTF_BSERR_2  22

#define xp_1            23
#define xp_2            24
```

```
#define xGAMSQ_1        25
#define xEPSI_1         26


#define sinf            (*(ptr + xSINF))
#define sinf_1          (*(ptr + xSINF_1))
#define P_sinf          (*(ptr + xSINF   - LQRDJPEF_CELL_SIZE))
#define P_sinf_1        (*(ptr + xSINF_1 - LQRDJPEF_CELL_SIZE))

#define sinb            (*(ptr + xSINB))
#define sinb_1          (*(ptr + xSINB_1))
#define P_sinb          (*(ptr + xSINB   - LQRDJPEF_CELL_SIZE))
#define P_sinb_1        (*(ptr + xSINB_1 - LQRDJPEF_CELL_SIZE))

#define cosf            (*(ptr + xCOSF))
#define cosf_1          (*(ptr + xCOSF_1))
#define P_cosf          (*(ptr + xCOSF   - LQRDJPEF_CELL_SIZE))
#define P_cosf_1        (*(ptr + xCOSF_1 - LQRDJPEF_CELL_SIZE))

#define cosb            (*(ptr + xCOSB))
#define cosb_1          (*(ptr + xCOSB_1))
#define P_cosb          (*(ptr + xCOSB   - LQRDJPEF_CELL_SIZE))
#define P_cosb_1        (*(ptr + xCOSB_1 - LQRDJPEF_CELL_SIZE))

#define epsf            (*(ptr + xEPSF))
#define P_epsf          (*(ptr + xEPSF   - LQRDJPEF_CELL_SIZE))

#define epsb            (*(ptr + xEPSB))
#define epsb_1          (*(ptr + xEPSB_1))
#define P_epsb          (*(ptr + xEPSB   - LQRDJPEF_CELL_SIZE))
#define P_epsb_1        (*(ptr + xEPSB_1 - LQRDJPEF_CELL_SIZE))

#define pief            (*(ptr + xPIEF))
#define pief_1          (*(ptr + xPIEF_1))
#define P_pief          (*(ptr + xPIEF   - LQRDJPEF_CELL_SIZE))
#define P_pief_1        (*(ptr + xPIEF_1 - LQRDJPEF_CELL_SIZE))

#define pieb            (*(ptr + xPIEB))
#define pieb_1          (*(ptr + xPIEB_1))
#define P_pieb          (*(ptr + xPIEB   - LQRDJPEF_CELL_SIZE))
#define P_pieb_1        (*(ptr + xPIEB_1 - LQRDJPEF_CELL_SIZE))


#define Fserr           (*(ptr + xFSERR))
#define Fserr_1         (*(ptr + xFSERR_1))
#define P_Fserr         (*(ptr + xFSERR   - LQRDJPEF_CELL_SIZE))
#define P_Fserr_1       (*(ptr + xFSERR_1 - LQRDJPEF_CELL_SIZE))
#define SQRTF_Fserr     (*(ptr + xSQRTF_FSERR))
#define SQRTF_Fserr_1   (*(ptr + xSQRTF_FSERR_1))
#define SQRTF_P_Fserr   (*(ptr + xSQRTF_FSERR   - LQRDJPEF_CELL_SIZE))
#define SQRTF_P_Fserr_1 (*(ptr + xSQRTF_FSERR_1 - LQRDJPEF_CELL_SIZE))

#define Bserr_1         (*(ptr + xBSERR_1))
#define Bserr_2         (*(ptr + xBSERR_2))
#define P_Bserr_1       (*(ptr + xBSERR_1 - LQRDJPEF_CELL_SIZE))
#define P_Bserr_2       (*(ptr + xBSERR_2 - LQRDJPEF_CELL_SIZE))
#define SQRTF_Bserr_1   (*(ptr + xSQRTF_BSERR_1))
#define SQRTF_Bserr_2   (*(ptr + xSQRTF_BSERR_2))
#define SQRTF_P_Bserr_1 (*(ptr + xSQRTF_BSERR_1 - LQRDJPEF_CELL_SIZE))
#define SQRTF_P_Bserr_2 (*(ptr + xSQRTF_BSERR_2 - LQRDJPEF_CELL_SIZE))
```

```
#define p_1             (*(ptr + xp_1))
#define p_2             (*(ptr + xp_2))
#define P_p_1           (*(ptr + xp_1     - LQRDJPEF_CELL_SIZE))
#define P_p_2           (*(ptr + xp_2     - LQRDJPEF_CELL_SIZE))

#define gams_1          (*(ptr + xGAMSQ_1))
#define P_gams_1        (*(ptr + xGAMSQ_1 - LQRDJPEF_CELL_SIZE))

#define epsi_1          (*(ptr + xEPSI_1))
#define P_epsi_1        (*(ptr + xEPSI_1  - LQRDJPEF_CELL_SIZE))

static FLOAT32
RSQRTF(
        FLOAT32 x) {

        return 1.0F / SQRTF(x);
}

/* ------------------------------------------------------------------------
   Name          : LQRDJPEF_Init
   Abstract      : Create a JPE context
   ------------------------------------------------------------------------ */

extern LQRDJPEF_Handle
LQRDJPEF_Init(
        INT32       NumCells,
        FLOAT32     Lambda,
        FLOAT32     SumErrInit,
        FLOAT32     GamsInit,
        FLOAT32     MinSumErr) {

        LQRDJPEF_Context    *jpe;

        jpe = malloc(sizeof(LQRDJPEF_Context));
        assert(jpe != NULL);

        jpe->NumCells = NumCells;
        jpe->Lambda = Lambda;
        jpe->SumErrInit = SumErrInit;
        jpe->GamsInit = GamsInit;
        jpe->MinSumErr = MinSumErr;
        jpe->SQRTF_Lambda = SQRTF(jpe->Lambda);
        jpe->SQRTF_SumErrInit = SQRTF(jpe->SumErrInit);

        jpe->cells = malloc(sizeof(LQRDJPEF_CELL) * (NumCells + 2));
        assert(jpe->cells != NULL);

        LQRDJPEF_Reset((LQRDJPEF_Handle)jpe);
        return ((LQRDJPEF_Handle)jpe);
}


/* ------------------------------------------------------------------------
   Name          : LQRDJPEF_Done
   Abstract      : Delete a JPE context
   ------------------------------------------------------------------------ */

extern VOID
LQRDJPEF_Done(
```

```
        LQRDJPEF_Handle     hJPE) {

        LQRDJPEF_Context    *jpe = (LQRDJPEF_Context *)hJPE;

        free(jpe->cells);
        free(jpe);
}

/* ------------------------------------------------------------------
   Name         : LQRDJPEF_Reset
   Abstract     : Reset a JPE context
   ------------------------------------------------------------------ */

extern VOID
LQRDJPEF_Reset(
        LQRDJPEF_Handle     hJPE) {

        LQRDJPEF_Context *jpe = (LQRDJPEF_Context *)hJPE;

        FLOAT32             *ptr;
        INT32               m;

        ptr = (FLOAT32 *)jpe->cells;
        for (m = 0; m <= jpe->NumCells; m++) {

        sinf    = 0.0F;
        sinb    = 0.0F;
        cosf    = 0.0F;
        cosb    = 0.0F;
        epsf    = 0.0F;
        epsb    = 0.0F;
        pief    = 0.0F;
        pieb    = 0.0F;
        p_1     = 0.0F;
        Fserr   = jpe->SumErrInit;
        Bserr_1 = jpe->SumErrInit;
        gams_1  = jpe->GamsInit;
        SQRTF_Fserr = jpe->SQRTF_SumErrInit;
        SQRTF_Bserr_1 = jpe->SQRTF_SumErrInit;

        ptr    += LQRDJPEF_CELL_SIZE;
        }
        ptr = (FLOAT32 *)jpe->cells;         /* Cell # 0 special case    */
        gams_1 = 1.0F;
}

/* ------------------------------------------------------------------
   Name         : LQRDJPEF_Calc
   Abstract     :
   ------------------------------------------------------------------ */

extern FLOAT32
LQRDJPEF_Calc(
        LQRDJPEF_Handle     hJPE,
        FLOAT32  nps,           /* noise plus signal          */
        FLOAT32  noise) {       /* noise reference            */

        LQRDJPEF_Context *jpe = (LQRDJPEF_Context *)hJPE;

        INT32       m;
```

```
        FLOAT32     *ptr;
        FLOAT32     tmp;

/*      Time update section                                          */

        ptr     = (FLOAT32 *)jpe->cells;

        for (m = 0; m <= jpe->NumCells; m++) {
        /* some of following delay elements are not needed */

        sinf_1  = sinf;
        sinb_1  = sinb;
        cosf_1  = cosf;
        cosb_1  = cosb;
        epsb_1  = epsb;
        pief_1  = pief;
        pieb_1  = pieb;

        Fserr_1 = Fserr;
        Bserr_2 = Bserr_1;
        p_2     = p_1;

        SQRTF_Bserr_2 = SQRTF_Bserr_1;
        SQRTF_Fserr_1 = SQRTF_Fserr;

        ptr += LQRDJPEF_CELL_SIZE;
        }

        /*  Order update section    */

        /*  Handle Cell # 0         */

        ptr = (FLOAT32 *)(jpe->cells); /* point to cell # 0 */
        epsf   = noise;
        epsb   = noise;
        epsi_1 = nps;

        /* rest of cells            */

        for (m = 1; m < jpe->NumCells; m++) {

            ptr += LQRDJPEF_CELL_SIZE;     /* access next cell */

            /* Prediction section */

            P_Bserr_1 = jpe->Lambda * P_Bserr_2 + P_epsb_1 * P_epsb_1;
            P_Bserr_1 = MAX(P_Bserr_1, jpe->MinSumErr);
            SQRTF_P_Bserr_1 = SQRTF(P_Bserr_1);
            tmp = RSQRTF(P_Bserr_1); /* this comes free on DSP */

            P_cosb_1  = jpe->SQRTF_Lambda * SQRTF_P_Bserr_2 * tmp;

            P_sinb_1  = P_epsb_1 * tmp;

            tmp = jpe->SQRTF_Lambda * P_pief_1;
            epsf      = P_cosb_1 * P_epsf - P_sinb_1 * tmp;

            P_pief    = P_cosb_1 * tmp + P_sinb_1 * P_epsf;

            gams_1    = P_cosb_1 * P_gams_1;
```

```
        P_Fserr    = jpe->Lambda * P_Fserr_1 + P_epsf * P_epsf;
        P_Fserr    = MAX(P_Fserr, jpe->MinSumErr);

        SQRTF_P_Fserr = SQRTF(P_Fserr);
        tmp = RSQRTF(P_Fserr); /* this comes free on DSP */

        P_cosf     = jpe->SQRTF_Lambda * SQRTF_P_Fserr_1 * tmp;

        P_sinf     = P_epsf * tmp;

        tmp = jpe->SQRTF_Lambda * P_pieb_1;
        epsb       = P_cosf * P_epsb_1 - P_sinf * tmp;

        P_pieb     = P_cosf * tmp + P_sinf * P_epsb_1;

        /* Joint Process Estimation section */

        tmp = jpe->SQRTF_Lambda * P_p_2;
        epsi_1     = P_cosb_1 * P_epsi_1 - P_sinb_1 * tmp;

        P_p_1      = P_cosb_1 * tmp + P_sinb_1 * P_epsi_1;
    }

    ptr += LQRDJPEF_CELL_SIZE;  /* access next cell */


    /* Do minimum work for JPE of very last cell
       only four equations are required for prediction section */

    P_Bserr_1 = jpe->Lambda * P_Bserr_2 + P_epsb_1 * P_epsb_1;
    P_Bserr_1 = MAX(P_Bserr_1, jpe->MinSumErr);

    SQRTF_P_Bserr_1 = SQRTF(P_Bserr_1);
    tmp = RSQRTF(P_Bserr_1); /* this comes free on DSP */

    P_cosb_1   = jpe->SQRTF_Lambda * SQRTF_P_Bserr_2 * tmp;

    P_sinb_1   = P_epsb_1 * tmp;

    /* Joint Process Estimation for last cell      */

    tmp = jpe->SQRTF_Lambda * P_p_2;
    epsi_1     = P_cosb_1 * P_epsi_1 - P_sinb_1 * tmp;

    P_p_1      = P_cosb_1 * tmp + P_sinb_1 * P_epsi_1;

    gams_1     = P_cosb_1 * P_gams_1;

    return(gams_1 * epsi_1);
}
```